Formalizing a Paraconsistent Logic in the Isabelle Proof Assistant

Villadsen, Jørgen; Schlichtkrull, Anders

Published in:
Transactions on Large-Scale Data- and Knowledge-Centered Systems

Link to article, DOI:
10.1007/978-3-662-55947-5_5

Publication date:
2017

Document Version
Peer reviewed version

Formalizing a Paraconsistent Logic in the Isabelle Proof Assistant

Jørgen Villadsen and Anders Schlichtkrull

DTU Compute, Technical University of Denmark, 2800 Kongens Lyngby, Denmark

Abstract. We present a formalization of a so-called paraconsistent logic that avoids the catastrophic explosiveness of inconsistency in classical logic. The paraconsistent logic has a countably infinite number of non-classical truth values. We show how to use the proof assistant Isabelle to formally prove theorems in the logic as well as meta-theorems about the logic. In particular, we formalize a meta-theorem that allows us to reduce the infinite number of truth values to a finite number of truth values, for a given formula, and we use this result in a formalization of a small case study.

Keywords: Paraconsistent Logic, Many-Valued Logic, Formalization, Isabelle Proof Assistant, Inconsistency, Paraconsistency

1 Introduction

Proof assistants are computer programs that assist users in conducting proofs. In general, proof assistants are useful tools both for clarifying concepts and for catching mistakes [14]. In addition, proof assistants are often able to perform calculations in different ways using rewriting rules or code generation. We use the Isabelle proof assistant [28,29], more precisely Isabelle's default higher-order logic called Isabelle/HOL, which includes powerful specification tools for advanced datatypes, inductive definitions and recursive functions.

1.1 Formalization in Proof Assistants

Today's proof assistants use proof systems with axioms and rules as famously characterized in the beginning of Kurt Gödel's seminal paper from 1931 on the Incompleteness Theorems [15]:

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

Modern computers are indeed excellent at following such mechanical rules used in proof systems. A book about "the seventeen provers of the world" included formalizations of a proof of the irrationality of $\sqrt{2}$ from researchers using various proof assistants and automatic theorem provers [48]. Arguably the two most used proof assistants with large mathematical libraries are Coq [4] and Isabelle [28,29] — both the results of more than 30 years of research in automated reasoning.

The creator of Isabelle, Lawrence C. Paulson, states in a recent paper on the first formalization of Gödel's Incompleteness Theorems [30]:

> Note that this paper contains no definitions or proofs as conventionally understood in mathematics; rather, it describes definitions and formal proofs that have been conducted in Isabelle/HOL, and lessons learned from them.

A formalization can catch mistakes, small or big, in definitions, theorems and proofs. Furthermore formalizations bring attention to vague specifications and make it easier to experiment with variants of definitions and theorems.

We return to the ins and outs of formalization in Isabelle in a moment.

1.2 Paraconsistency

In brief, paraconsistency is about handling contradictions in a coherent way, and many approaches have been investigated [1, 2, 9, 11, 31, 32, 46]. In classical logic there are only two truth values and everything follows from a contradiction, but in a paraconsistent logic not everything follows from a contradiction.

In the present paper we formalize the syntax and semantics of a many-valued paraconsistent logic with a countably infinite number of truth values [20, 40–43]. We do not consider any proof systems for our particular paraconsistent logic, but we can prove theorems and non-theorems using the semantics like it is done with truth tables for classical propositional logic. However, since our paraconsistent logic has infinitely many truth values, it is far from obvious that finite truth tables suffice.

Although we in the present paper formalize a particular many-valued paraconsistent logic, the logic can be changed or even replaced in the formalization. Isabelle would then show which formal theorems and proofs need to be adapted.

It is helpful to distinguish between weak and strong paraconsistency, quoting Weber [46]:

> Roughly, weak paraconsistency is the cluster concept that
> - any apparent contradictions are always due to human error;
> - classical logic is preferable, and in a better world where humans did not err, we would use classical logic;
> - no true theory would ever contain an inconsistency.

This is our view on the matter, however, there is another view, again quoting Weber [46]:

> On the other side, strong paraconsistency includes ideas like

- Some contradictions may not be errors;
- classical logic is wrong in principle;
- some true theories may actually be inconsistent.

The proof assistant Isabelle uses classical logic and it seems hard to adhere to strong paraconsistency then.

The standard definition of paraconsistency is in terms of non-explosion [46]:

> A logic is paraconsistent iff it is not the case for all sentences A, B that $A, \neg A \vdash B$.

However, in our paraconsistent logic we have nothing on the left-hand side of the turnstile ($\vdash$) so we instead consider the following statement:

$$\vdash A \wedge \neg A \rightarrow B$$

In order to illustrate the notion of entailment we introduce a small case study. Classical logic is problematic in, for example, multi-agent systems, since the belief base of an agent very well could contain contradictory beliefs and thus be inconsistent. For example, as a small case study, consider an agent with a set of atomic beliefs (item 0) and a few simple rules:

0. $P \wedge Q \wedge \neg R$
1. $P \wedge Q \rightarrow R$
2. $R \rightarrow S$

This leaves the agent with contradictory beliefs, namely R and $\neg R$, so the agent might start behaving in an undesirable way if it uses classical logic. It could now believe that $\neg P$, or $\neg Q$ — or even φ for any formula φ. Using our paraconsistent logic this is not the case [20]. We return to the case study in Section 8.

In multi-agent systems where agents have to take into account the beliefs of other agents, it can be difficult to use other approaches like belief revision [17] because belief revision seems to be a rather strong assumption about the capabilities of other agents whereas our many-valued paraconsistent logic is "just" a generalization of classical logic with respect to both syntax (new operators) and semantics (more truth values). Think of a judge who has conflicting arguments of the prosecutor and the defender of a culprit. Such a reasoner needs to take an unbiased, impartial point of view without the possibility of coercing neither the prosecutor nor the defender to change their belief in favor of the counterparty.

1.3 Formalization of Logic

We formally prove theorems in the logic as well as theorems about the logic. The proofs are checked by the Isabelle proof assistant [28, 29]. By submission to the online Archive of Formal Proofs we make sure that the proofs are maintained continuously against the current stable release of Isabelle [37]:

`http://isa-afp.org/browser_info/current/AFP/Paraconsistency`

The above link provides PDF documents with or without proofs and the theory file can be browsed online. The Archive of Formal Proofs has mid 2017 almost 100,000 theorems and lemmas in total and covers numerous advanced topics in mathematics, logic and computer science:

`http://isa-afp.org/statistics.shtml`

Since the start in 2004 more than 250 authors have contributed. There are 42 entries in the logic category. For example, Paulson's formalization of Gödel's Incompleteness Theorems is in the Archive of Formal Proofs [30] and so are two recent formalizations of proof systems:

1. Jensen, Schlichtkrull and Villadsen [19] formalize a declarative first-order prover with equality based on John Harrison's *Handbook of Practical Logic and Automated Reasoning* and the entire prover can be executed within Isabelle as a very simple interactive proof assistant.
2. Michaelis and Nipkow [25] formalize proof systems for classical propositional logic and prove the most important meta-theoretic results about semantics and proofs: compactness, soundness, completeness, translations between proof systems, cut-elimination, interpolation and model existence.

These formalizations as well as our work on paraconsistency are in the repository IsaFoL, Isabelle Formalization of Logic, with the goal to develop lemma libraries and methodology for formalizing modern research in automated reasoning:

`https://bitbucket.org/isafol/isafol/`

The repository gives an overview of recent formalizations of logics in the Isabelle proof assistant. A state-of-the-art approach to the formalization of soundness and completeness results for logics has been developed by Blanchette, Popescu and Traytel [6] and the formalization is available in the Archive of Formal Proofs, but paraconsistent and/or many-valued logics are not considered.

1.4 The Isabelle Proof Assistant

One of Isabelle's central components is the Isar language for writing proofs [47]. The language bears resemblance to logical systems, handwritten mathematical proofs and programming languages.

It is similar to logical systems, in particular natural deduction, in that formulas can be proved by breaking them down into smaller parts using appropriate inference rules.

It is similar to mathematical paper proofs because an Isar-proof can be written as a sequence of sentences, each one following from the previous ones, that leads us towards a goal. In particular it is very similar to the structured proof style that Lamport [22, 23] recommends for the 21st century.

It is similar to a programing language in that its syntax is structured and consists of various commands — these commands instruct Isabelle on how to prove the desired theorems.

Another important feature of Isar is that it allows one to mix this structured reasoning with state-of-the-art automatic theorem provers.

We illustrate the language with a simple proof of a — perhaps — surprising theorem called the drinker's paradox. The theorem states that in a bar there is a person such that if he is drinking then everybody is drinking (we use predicate D for drinking). We have the following Isar proof:

```
theorem "∃x. D x ⟶ (∀x. D x)"
proof cases
  assume "∀x. D x"
  then have "P ⟶ (∀x. D x)" for P ..
  then show ?thesis ..
next
  assume "¬ (∀x. D x)"
  then have "∃x. ¬ D x" by simp
  then obtain a where nda: "¬ D a" ..
  have "D a ⟶ (∀x. D x)"
  proof
    assume "D a"
    then show "∀x. D x" using nda by metis
  qed
  then show ?thesis ..
qed
```

Even for the uninitiated the proof should be at least somewhat readable because keywords such as **theorem**, **proof**, **assume**, **then** and **have** are well known from mathematical literature. Furthermore, each sentence is written in Isabelle/HOL, which has a similar notation to e.g. first-order logic (FOL).

Let us describe the Isar-proof in detail. After we state the theorem comes a proof block starting with **proof** *cases* and ending with the **qed** on the very last line. This proof block allows us to do proof by cases on whether $\forall x.D\ x$ is true or not and in both cases we are obliged to prove the theorem. One can easily imagine a classical proof system with such a rule.

We start by proving the first case $\forall x.D\ x$. This proof starts with **assume** $\forall x.D\ x$ and ends with **then show** *?thesis* **..** two lines below. The proof is similar to a paper proof of a sequence of three sentences – each line corresponding to a sentence. Here *?thesis* refers to the theorem we are proving.

Next comes the second case $\neg(\forall x.D\ x)$. Again the proof is a sequence of sentences. To convince Isabelle that the first sentence follows from the second,

we apply the proof method called *simp*, which does simplifications, by writing **by** *simp*. Next we use the **obtain** command to obtain the element a that the previous sentence proved exists. After this we prove an implication using an inner proof block. Notice how the inner proof block is nested in the outer proof block. In this inner proof block we prove the implication by breaking it down structurally using the implication introduction rule from natural deduction, which states that to prove an implication we assume the antecedent and prove the consequent. Notice also that in the inner proof block we use another proof method called *metis* and additionally allow it to use the previously labelled sentence *nda*. The *metis* proof method is an automatic theorem prover.

Note that Isabelle/HOL is written in a curried style. This means that function application is written without parentheses unless necessary. An example is $D\ x$ as we saw above. Additionally n-argument functions are typically given a type

$$'a_1 \Rightarrow ('a_2 \Rightarrow ('a_3 \Rightarrow (\cdots \Rightarrow ('a_n \Rightarrow 'b)\cdots)))$$

or, if we drop the parentheses

$$'a_1 \Rightarrow 'a_2 \Rightarrow 'a_3 \Rightarrow \cdots \Rightarrow 'a_n \Rightarrow 'b$$

instead of

$$('a_1 \times \cdots \times 'a_n) \Rightarrow 'b.$$

Therefore, an application of, e.g., a binary function R to two arguments x and y is written as $R\ x\ y$.

You can try to write the above proof in Isabelle. You will notice that it is easy to accidentally introduce some mistake that makes Isabelle unable to finish the proof. This is the advantage of proving theorems in Isabelle — the system is very good at catching small and big mistakes.

1.5 Contributions and Overview

All formulas in the present paper have been checked by the Isabelle proof assistant except for the informal presentation in Section 2. We must emphasize that the proofs in the paper are not generated by Isabelle or any other computer program. All proofs in the paper are word for word authored by us. Our proofs can — at least in principle — be read and checked by other Isabelle users and can also be read and checked by the Isabelle proof assistant — and have therefore been accepted for the Archive of Formal Proofs.

Our main contributions are as follows.

- In Section 3: A formalization of the syntax and semantics of the many-valued paraconsistent logic with many new definitions.
- In Section 4 and Section 5: A series of theorems and non-theorems of which only a few have been considered in our previous publications.

– In Section 6: A new analysis of the required number of truth values for counterexamples.

– In Section 7: A reduction theorem that was originally mentioned without proof in our extended abstract [20].

– In Section 8: A proposal for entailment and verification of the results for the case study presented in the present section — these results were also mentioned without proof in our extended abstract [20].

We describe related work in Section 9 and conclude in Section 10.

2 The Paraconsistent Logic — An Informal Presentation

By "informal" we here mean that we provide a mathematical presentation of the logic but the formalization in the Isabelle proof assistant is provided in the following sections. We describe the propositional fragment of our higher-order many-valued paraconsistent logic [43]. We follow the concise presentation in our extended abstract [20] but with some additional abbreviations.

2.1 Semantic Clauses and Key Equalities

We have the two classical determinate truth values $\{\bullet, \circ\}$ for truth and falsity and a countably infinite set of indeterminate truth values $\{^{\mathrm{I}}, ^{\mathrm{II}}, ^{\mathrm{III}}, \ldots\}$.

The indeterminate truth values are not ordered with respect to truth content. The only designated truth value is $\bullet$ and hence only this truth value yields the logical truths.

This use of $\bullet$ and $\circ$ for the classical truth values goes back to our previous publications [41–43] and the references therein. Note that as usual the corresponding operators are $\top$ and $\bot$ (see below).

The logic is a generalization of Łukasiewicz's three-valued logic — originally proposed 1920–30 — with the intermediate value duplicated many times and ordered such that none of the copies of this value imply other ones, but the logic differs from Łukasiewicz's many-valued logics as well as from logics based on bilattices [16].

The motivation for the logical operators is based on key equalities shown to the right of the semantic clauses. We also have $\varphi \Leftrightarrow \neg\neg\varphi$ as a key equality. Negation does not change indeterminate truth values since they are not ordered with respect to truth content. In the higher-order paraconsistent logic [41–43] the key equalities are proper equalities $=$ corresponding to $\Leftrightarrow$ here. The key equalities do not provide an axiomatization as such but rather they provide for each logical operator the semantic clauses except for the default case.

Note that in the semantic clauses several cases may apply if and only if they agree on the result and that the semantic clauses work for classical logic too. Atoms are interpreted by the basic semantic clause and $\top$ by $[\![\top]\!] = \bullet$.

$$[\![\neg\varphi]\!] \;=\; \begin{cases} \bullet & \text{if } [\![\varphi]\!] = \circ \qquad \top \;\Leftrightarrow\; \neg\bot \\ \circ & \text{if } [\![\varphi]\!] = \bullet \qquad \bot \;\Leftrightarrow\; \neg\top \\ [\![\varphi]\!] & \text{otherwise} \end{cases}$$

$$[\![\varphi \wedge \psi]\!] \;=\; \begin{cases} [\![\varphi]\!] & \text{if } [\![\varphi]\!] = [\![\psi]\!] \qquad \varphi \;\Leftrightarrow\; \varphi \wedge \varphi \\ [\![\psi]\!] & \text{if } [\![\varphi]\!] = \bullet \qquad \psi \;\Leftrightarrow\; \top \wedge \psi \\ [\![\varphi]\!] & \text{if } [\![\psi]\!] = \bullet \qquad \varphi \;\Leftrightarrow\; \varphi \wedge \top \\ \circ & \text{otherwise} \end{cases}$$

Abbreviations:

$$\bot \;\equiv\; \neg\top \qquad \varphi \vee \psi \;\equiv\; \neg(\neg\varphi \wedge \neg\psi)$$

We continue with biimplication (and we then simply obtain implication and modality as abbreviations). The semantic clauses for $\leftrightarrow$ extend the clauses for $\Leftrightarrow$, which always give a determinate truth value.

$$[\![\varphi \Leftrightarrow \psi]\!] \;=\; \begin{cases} \bullet & \text{if } [\![\varphi]\!] = [\![\psi]\!] \\ \circ & \text{otherwise} \end{cases}$$

$$[\![\varphi \leftrightarrow \psi]\!] \;=\; \begin{cases} \bullet & \text{if } [\![\varphi]\!] = [\![\psi]\!] \qquad \top \;\Leftrightarrow\; \varphi \leftrightarrow \varphi \\ [\![\psi]\!] & \text{if } [\![\varphi]\!] = \bullet \qquad \psi \;\Leftrightarrow\; \top \leftrightarrow \psi \\ [\![\varphi]\!] & \text{if } [\![\psi]\!] = \bullet \qquad \varphi \;\Leftrightarrow\; \varphi \leftrightarrow \top \\ [\![\neg\psi]\!] & \text{if } [\![\varphi]\!] = \circ \qquad \neg\psi \;\Leftrightarrow\; \bot \leftrightarrow \psi \\ [\![\neg\varphi]\!] & \text{if } [\![\psi]\!] = \circ \qquad \neg\varphi \;\Leftrightarrow\; \varphi \leftrightarrow \bot \\ \circ & \text{otherwise} \end{cases}$$

Abbreviations:

$$\Box\varphi \;\equiv\; \varphi \;\Leftrightarrow\; \top \qquad \neg\varphi \;\equiv\; \Box\neg\varphi \qquad \nabla\varphi \;\equiv\; \neg\Box(\varphi \vee \neg\varphi)$$

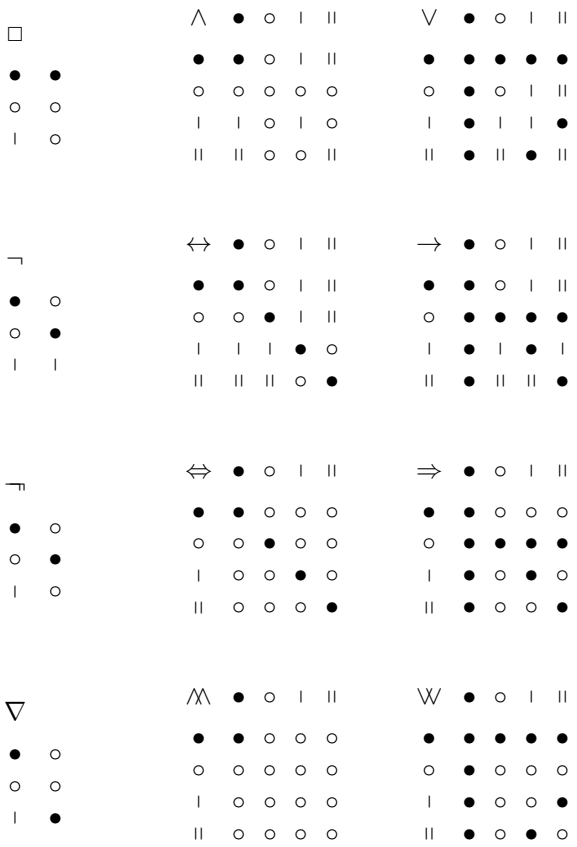
$$\varphi \Rightarrow \psi \;\equiv\; \varphi \;\Leftrightarrow\; \varphi \wedge \psi \qquad \varphi \rightarrow \psi \;\equiv\; \varphi \leftrightarrow \varphi \wedge \psi$$

$$\varphi \mathbin{\mathbb{M}} \psi \;\equiv\; \Box(\varphi \wedge \psi) \qquad \varphi \mathbin{\mathbb{W}} \psi \;\equiv\; \Box(\varphi \vee \psi)$$

2.2 Truth Tables

Although we have a countably infinite set of truth values we can investigate the logic by truth tables since the indeterminate truth values are not ordered with respect to truth content.

In order to grasp the main properties of the operators we need just the two indeterminate truth values ı and ıı as in the following truth tables.



Observe that with respect to validity, viz. the classical determinate truth value • for truth, the operators ¬ and ¬╕ behave the same and likewise for the operators ⇔ and ↔ and ⇒ and →, respectively.

The truth tables are obtained from the semantic clauses. The formalization includes features for this calculation but since the truth tables are solely for informal presentation purposes we have typeset them using the same symbols as used in the semantics clauses. However, Section 6 contains a few truth tables calculated by Isabelle. The theory file for the formalization includes all calculated truth tables.

3 Syntax and Semantics

We formalize the syntax and semantics of the many-valued paraconsistent logic as follows. For the syntax we first define the propositional symbols (*id*) as a simple abbreviation for text strings and the formulas (*fm*) as a recursive datatype (almost as the productions for a context-free grammar).

type-synonym *id* = *string*

datatype *fm* =
Pro id |
Truth |
Neg′ fm |
Con′ fm fm |
Eql fm fm |
Eql′ fm fm

We then define the remaining operators as abbreviations. We do this with Isabelle's **abbreviation** command by giving the name of the abbreviated operator, e.g. *Falsity*, and thereafter its type, e.g. *fm*. After the **where** keyword we write the equality that defines the abbreviation, e.g. *Falsity* ≡ *Neg′ Truth*.

abbreviation *Falsity* :: *fm* **where** *Falsity* ≡ *Neg′ Truth*

abbreviation *Dis′* :: *fm* ⇒ *fm* ⇒ *fm*
where *Dis′ p q* ≡ *Neg′* (*Con′* (*Neg′ p*) (*Neg′ q*))

abbreviation *Imp* :: *fm* ⇒ *fm* ⇒ *fm* **where** *Imp p q* ≡ *Eql p* (*Con′ p q*)

abbreviation *Imp′* :: *fm* ⇒ *fm* ⇒ *fm* **where** *Imp′ p q* ≡ *Eql′ p* (*Con′ p q*)

abbreviation *Box* :: *fm* ⇒ *fm* **where** *Box p* ≡ *Eql p Truth*

abbreviation *Neg* :: *fm* ⇒ *fm* **where** *Neg p* ≡ *Box* (*Neg′ p*)

abbreviation *Con* :: *fm* ⇒ *fm* ⇒ *fm* **where** *Con p q* ≡ *Box* (*Con′ p q*)

abbreviation *Dis* :: *fm* ⇒ *fm* ⇒ *fm* **where** *Dis p q* ≡ *Box* (*Dis′ p q*)

abbreviation *Cla* :: *fm* ⇒ *fm* **where** *Cla p* ≡ *Dis* (*Box p*) (*Eql p Falsity*)

abbreviation *Nab* :: *fm* ⇒ *fm* **where** *Nab p* ≡ *Neg* (*Cla p*)

The truth values are the two determinate truth values and the countably infinite number of indeterminate truth values. We also define a useful abbreviation for negation (*eval-neg*). This function turns *Det False* into *Det True* and vice versa, but does not change the value of indeterminate truth values. The function is defined by a case-expression that matches x with the patterns *Det False*, *Det True* and *Indet n* (where n can be any value), and returns the value to the

right of the corresponding arrow. Finally, we define the semantics as a recursive function on the structure of the given formula (*eval*) using the **fun** command. The function is defined by a number of equations. In one of the equations we again use a case expression. This time the case expression contains a number of dummy variables (wildcard patterns), which are typeset as dashes (-). Each one of these will independently match with anything.

```
datatype tv = Det bool | Indet nat

abbreviation (input) eval-neg :: tv ⇒ tv
where
  eval-neg x ≡
    (
      case x of
        Det False ⇒ Det True |
        Det True ⇒ Det False |
        Indet n ⇒ Indet n
    )

fun eval :: (id ⇒ tv) ⇒ fm ⇒ tv
where
  eval i (Pro s) = i s |
  eval i Truth = Det True |
  eval i (Neg' p) = eval-neg (eval i p) |
  eval i (Con' p q) =
    (
      if eval i p = eval i q then eval i p else
      if eval i p = Det True then eval i q else
      if eval i q = Det True then eval i p else Det False
    ) |
  eval i (Eql p q) =
    (
      if eval i p = eval i q then Det True else Det False
    ) |
  eval i (Eql' p q) =
    (
      if eval i p = eval i q then Det True else
        (
          case (eval i p, eval i q) of
            (Det True, -) ⇒ eval i q |
            (-, Det True) ⇒ eval i p |
            (Det False, -) ⇒ eval-neg (eval i q) |
            (-, Det False) ⇒ eval-neg (eval i p) |
            - ⇒ Det False
        )
    )
```

We prove a few useful results about the semantics. We first prove a formulation of the semantics for *Eql'* and *Neg'* without the *eval-neg* abbreviation. We then prove that a double negation with *Neg'* does not change the semantics.

```
theorem eval-equality:
  eval i (Eql' p q) =
    (
      if eval i p = eval i q then Det True else
      if eval i p = Det True then eval i q else
      if eval i q = Det True then eval i p else
      if eval i p = Det False then eval i (Neg' q) else
      if eval i q = Det False then eval i (Neg' p) else
      Det False
    )
  by (cases eval i p; cases eval i q) simp-all

theorem eval-negation:
  eval i (Neg' p) =
    (
      if eval i p = Det False then Det True else
      if eval i p = Det True then Det False else
      eval i p
    )
  by (cases eval i p) simp-all

corollary eval i (Cla p) = eval i (Box (Dis' p (Neg' p)))
  using eval-negation
  by simp

lemma double-negation: eval i p = eval i (Neg' (Neg' p))
  using eval-negation
  by simp
```

We define the notion of valid formulas by quantifying over all interpretations.

```
definition valid :: fm ⇒ bool
where
  valid p ≡ ∀ i. eval i p = Det True

proposition valid Truth and ¬ valid Falsity
  unfolding valid-def
  by simp-all
```

The last proposition shows that the logic is consistent in the sense that there is a formula which is a theorem and not all formulas are theorems. The proof is explained in the next section.

4 Various Theorems and Proof Styles

We prove a series of theorems and non-theorems most of which are schemata. The purpose of the following quite long list is twofold: to investigate our paraconsistent logic, and secondly, to show a number of proof styles.

The first seven propositions are proved by unfolding the definition of validity and then simplifying the result to a true atomic proposition. The next two propositions are proved by the *metis* proof method as explained in Section 1. We let *metis* use a number of lemmas including *eval-equality*, *eval-negation* and two lemmas about respectively truth values and evaluation, that Isabelle proved implicitly when we defined these notions. Isabelle's powerful Sledgehammer tool has been used to obtain the proofs [5].

The next proposition — P is not valid — is proved by the *auto* proof method, which does a combination of simplification and classical reasoning. The following proposition — $\neg P$ is not valid — is proved by manually providing a counterexample – unfortunately Sledgehammer cannot find a proof for this proposition. The counterexample is the interpretation that maps everything to *True*. It is written as a λ-expression as known from the λ-calculus. In general, a λ-expression $\lambda x.\, F\, x$ represents the function that takes any x as input and returns $F\, x$.

Hereafter comes a proposition stating that the validity of p implies the non-validity of *Neg' p*. This is written using keywords **assumes** and **shows**, which logically is the same as if we had explicitly written an implication $\longrightarrow$, but will make theorems easier to read when there are many assumptions. Several of the following propositions are written in the same style.

The remaining propositions are proved using more or less the same proof methods (one proposition requires the so-called *force* proof method that can prove some propositions where *auto* gives up).

Some propositions have assumptions and in the proof the special fact *assms* can be used to refer to the assumptions.

In Isabelle there is no technical difference between the keywords **theorem**, **corollary**, **proposition** and **lemma**. We have found it useful to always name theorems and simply take propositions to be unnamed theorems. Lemmas are stepping stones and must of course have names in order to be used later in proofs. A corollary is taken to be readily proved from a theorem; see the theorem named *conjunction* (after 14 propositions).

```
proposition valid (Cla (Box p)) and ¬ valid (Nab (Box p))
  unfolding valid-def
  by simp-all

proposition valid (Cla (Cla p)) and ¬ valid (Nab (Nab p))
  unfolding valid-def
  by simp-all

proposition valid (Cla (Nab p)) and ¬ valid (Nab (Cla p))
  unfolding valid-def
  by simp-all

proposition valid (Box p) ⟷ valid (Box (Box p))
  unfolding valid-def
  by simp
```

```
proposition valid (Neg p) ⟷ valid (Neg' p)
  unfolding valid-def
  by simp

proposition valid (Con p q) ⟷ valid (Con' p q)
  unfolding valid-def
  by simp

proposition valid (Dis p q) ⟷ valid (Dis' p q)
  unfolding valid-def
  by simp

proposition valid (Eql p q) ⟷ valid (Eql' p q)
  unfolding valid-def
  using eval.simps tv.inject eval-equality eval-negation
  by (metis (full-types))

proposition valid (Imp p q) ⟷ valid (Imp' p q)
  unfolding valid-def
  using eval.simps tv.inject eval-equality eval-negation
  by (metis (full-types))

proposition ¬ valid (Pro ''p'')
  unfolding valid-def
  by auto

proposition ¬ valid (Neg' (Pro ''p''))
proof −
  have eval (λs. Det True) (Neg' (Pro ''p'')) = Det False
    by simp
  then show ?thesis
    unfolding valid-def
    using tv.inject
    by metis
qed

proposition assumes valid p shows ¬ valid (Neg' p)
  using assms
  unfolding valid-def
  by simp

proposition assumes valid (Neg' p) shows ¬ valid p
  using assms
  unfolding valid-def
  by force

proposition valid (Neg' (Neg' p)) ⟷ valid p
  unfolding valid-def
  using double-negation
  by simp
```

```
theorem conjunction: valid (Con' p q) ⟷ valid p ∧ valid q
  unfolding valid-def
  by auto

corollary assumes valid (Con' p q) shows valid p and valid q
  using assms conjunction
  by simp-all

proposition assumes valid p and valid (Imp p q) shows valid q
  using assms eval.simps tv.inject
  unfolding valid-def
  by (metis (full-types))

proposition assumes valid p and valid (Imp' p q) shows valid q
  using assms eval.simps tv.inject eval-equality
  unfolding valid-def
  by (metis (full-types))
```

The key equalities from Section 2 can also be proved but they are omitted here. The theory file for the formalization has the details.

The preceding propositions show that our paraconsistent logic is well-behaved in many ways. For example, the last propositions prove the rule of modus ponens for both kinds of implication.

5 Counterexamples for Non-Theorems

We introduce the possibility for restricting the domain of truth values and use it for stating counterexamples.

We first define a function *domain* that, for any given set of natural numbers, constructs the corresponding domain in our logic. It does so by turning the natural numbers in the set into indeterminate truth values using *Indet* and additionally adding the determinate truth values to the set.

We then prove the theorem *universal-domain* where the first universal set $\{n.\,True\}$ has type *nat set* and the second universal set $\{x.\,True\}$ has type *tv set*. The function *domain* provides the correspondence.

```
definition domain :: nat set ⇒ tv set
where
  domain U ≡ {Det True, Det False} ∪ Indet ‘ U

theorem universal-domain: domain {n. True} = {x. True}
proof −
  have ∀ x. x = Det True ∨ x = Det False ∨ x ∈ range Indet
    using range-eqI tv.exhaust tv.inject
    by metis
  then show ?thesis
    unfolding domain-def
```

```
    by blast
qed
```

We define the notion of valid formulas restricted to a given set of indeterminate truth values. We say that a formula p is valid in U if it is valid considering not all indeterminate truth values, but only those from U. Or more precisely, if p always evaluates to true in any interpretation i that has *domain U* as function range. In the formalization we use Isabelle/HOL's *range* function to get the range of i.

```
definition valid-in :: nat set ⇒ fm ⇒ bool
where
  valid-in U p ≡ ∀ i. range i ⊆ domain U ⟶ eval i p = Det True

abbreviation valid-boole :: fm ⇒ bool where valid-boole p ≡ valid-in {} p

proposition valid p ⟷ valid-in {n. True} p
  unfolding valid-def valid-in-def
  using universal-domain
  by simp

theorem valid-valid-in: assumes valid p shows valid-in U p
  using assms
  unfolding valid-in-def valid-def
  by simp

theorem transfer: assumes ¬ valid-in U p shows ¬ valid p
  using assms valid-valid-in
  by blast
```

In particular the above theorem (*transfer*) is useful in order to prove that a formula is not valid. As a particular example we will show that $P \wedge \neg P \rightarrow Q$ is not valid. First we show that it is valid in the boolean logic. Next we show that it is not valid in *domain* $\{1\}$. We do this by providing a counterexample. With the **let** command we define the counterexample *?i* (the **let** command requires that we have this question mark). The counterexample is first defined as returning *Indet 1* on any input, and is then modified (using :=) to return *Det False* on input q. The proof uses the **moreover** and **ultimately** commands. This works in the way that the statements just before each of the moreover commands are collected and used to prove the statement after the **ultimately** command. After proving this result, we use it together with *transfer* to prove that the formula is not valid.

```
abbreviation (input) Explosion :: fm ⇒ fm ⇒ fm
where
  Explosion p q ≡ Imp' (Con' p (Neg' p)) q

proposition valid-boole (Explosion (Pro ''p'') (Pro ''q''))
  unfolding valid-in-def
proof (rule; rule)
```

```
  fix i :: id ⇒ tv
  assume range i ⊆ domain {}
  then have
      i ''p'' ∈ {Det True, Det False}
      i ''q'' ∈ {Det True, Det False}
    unfolding domain-def
    by auto
  then show eval i (Explosion (Pro ''p'') (Pro ''q'')) = Det True
    by (cases i ''p''; cases i ''q'') simp-all
qed

lemma explosion-counterexample:
    ¬ valid-in {1} (Explosion (Pro ''p'') (Pro ''q''))
proof −
  let ?i = (λs. Indet 1)(''q'' := Det False)
  have range ?i ⊆ domain {1}
    unfolding domain-def
    by (simp add: image-subset-iff)
  moreover have eval ?i (Explosion (Pro ''p'') (Pro ''q'')) = Indet 1
    by simp
  moreover have Indet 1 ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid-in-def
    by metis
qed

theorem explosion-not-valid: ¬ valid (Explosion (Pro ''p'') (Pro ''q''))
  using explosion-counterexample transfer
  by simp
```

The last theorem shows that the many-valued logic is a paraconsistent logic since $P \wedge \neg P \rightarrow Q$ is not valid.

6 On the Number of Truth Values

For the normal two-value boolean propositional logic we can decide if a formula is valid or not by enumerating all interpretations and checking if they satisfy our formula. This approach will clearly not work for our many-valued logic since there are infinitely many truth values and thus also infinitely many interpretations.

However, it turns out that we do not need to consider all possibilities of truth values. For any formula there is a finite subset that it suffices to check. In this section we will argue for a lower bound on the size of this subset. Specifically we will argue that for an arbitrary formula containing n different propositional symbols, we need to consider interpretations with n different indeterminate truth values.

We first consider the simple case of formulas with one propositional symbol. In order to conduct the analysis, we first prove that if the range of an inter-

pretation is a subset of *domain U*, then any formula will evaluate to a value in *domain U* under this interpretation. Then we consider the example of *Cla p* where *Cla* is the unary operator that evaluates to true when its operand evaluates to a *Cla*ssical truth value. Let us introduce the informal notation Δ for *Cla*. We prove that Δp is valid in all boolean interpretations. Next we prove that it is not valid in *domain* {1}. Therefore we can conclude that considering 0 indeterminate truth values is not enough – we need to consider at least 1. We have also printed its truth table for illustration. In the calculated truth table below * is used for • and *o* is used for ∘. The functions *unary* and *binary* return truth tables as strings for unary and binary operators, respectively, and the theory file for the formalization has the details (about 50 lines of code, cf. [45]). The proof method *code-simp* performs the calculations of the truth table strings.

```
lemma ranges: assumes range i ⊆ domain U shows eval i p ∈ domain U
  using assms
  unfolding domain-def
  by (induct p) auto

proposition
  unary (Cla (Pro ''p'')) [Det True, Det False, Indet 1] = ''
  *
  *
  o
''
  by code-simp

proposition valid-boole (Cla p)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range i ⊆ domain {}
  then have
      eval i p ∈ {Det True, Det False}
    using ranges[of i {}]
    unfolding domain-def
    by auto
  then show eval i (Cla p) = Det True
    by (cases eval i p) simp-all
qed

proposition ¬ valid-in {1} (Cla (Pro ''p''))
proof −
  let ?i = λs. Indet 1
  have range ?i ⊆ domain {1}
    unfolding domain-def
    by (simp add: image-subset-iff)
  moreover have eval ?i (Cla (Pro ''p'')) = Det False
    by simp
  moreover have Det False ≠ Det True
```

```
    by simp
  ultimately show ?thesis
    unfolding valid-in-def
    by metis
qed
```

We repeat the exercise for a formula with two propositional symbols. This time we consider the formula Δ_2, which is $(\Delta p \vee \Delta q) \vee (p \Leftrightarrow q)$. We prove it valid in all boolean interpretations as well as in all interpretations with *domain* $\{1\}$. Next we prove that it is not valid in *domain* $\{1, 2\}$. Therefore it is not enough to consider 1 indeterminate truth value – we need to consider at least 2.

```
abbreviation (input) Cla2 :: fm ⇒ fm ⇒ fm
where
  Cla2 p q ≡ Dis (Dis (Cla p) (Cla q)) (Eql p q)

proposition
  binary (Cla2 (Pro ''p'') (Pro ''q''))
      [Det True, Det False, Indet 1, Indet 2] = ''
  ****
  ****
  ***o
  **o*
''
  by code-simp

proposition valid-boole (Cla2 p q)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range: range i ⊆ domain {}
  then have
      eval i p ∈ {Det True, Det False}
      eval i q ∈ {Det True, Det False}
    using ranges[of i {}]
    unfolding domain-def
    by auto
  then show eval i (Cla2 p q) = Det True
    by (cases eval i p; cases eval i q) simp-all
qed

proposition valid-in {1} (Cla2 p q)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range: range i ⊆ domain {1}
  then have
      eval i p ∈ {Det True, Det False, Indet 1}
      eval i q ∈ {Det True, Det False, Indet 1}
    using ranges[of i {1}]
```

```
    unfolding domain-def
    by auto
  then show eval i (Cla2 p q) = Det True
    by (cases eval i p; cases eval i q) simp-all
qed

proposition ¬ valid-in {1, 2} (Cla2 (Pro ''p'') (Pro ''q''))
proof −
  let ?i = (λs. Indet 1)(''q'' := Indet 2)
  have range ?i ⊆ domain {1, 2}
    unfolding domain-def
    by (simp add: image-subset-iff)
  moreover have eval ?i (Cla2 (Pro ''p'') (Pro ''q'')) = Det False
    by simp
  moreover have Det False ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid-in-def
    by metis
qed
```

We repeat the exercise for a formula with three propositional symbols. This time we consider the formula Δ_3, which is $(\Delta p \vee \Delta q \vee \Delta r) \vee ((p \Leftrightarrow q) \vee (p \Leftrightarrow r) \vee (q \Leftrightarrow r))$. We prove it valid in all boolean interpretations as well as in all interpretations with *domain* $\{1\}$ and *domain* $\{1, 2\}$. Next we prove that it is not valid in *domain* $\{1, 2, 3\}$. Therefore it is not enough to consider 2 indeterminate truth values – we need to consider at least 3.

```
abbreviation (input) Cla3 :: fm ⇒ fm ⇒ fm ⇒ fm
where
  Cla3 p q r ≡ Dis (Dis (Cla p) (Dis (Cla q) (Cla r)))
     (Dis (Eql p q) (Dis (Eql p r) (Eql q r)))

proposition valid-boole (Cla3 p q r)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range i ⊆ domain {}
  then have
      eval i p ∈ {Det True, Det False}
      eval i q ∈ {Det True, Det False}
      eval i r ∈ {Det True, Det False}
    using ranges[of i {}]
    unfolding domain-def
    by auto
  then show eval i (Cla3 p q r) = Det True
    by (cases eval i p; cases eval i q; cases eval i r) simp-all
qed

proposition valid-in {1} (Cla3 p q r)
```

```
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range i ⊆ domain {1}
  then have
      eval i p ∈ {Det True, Det False, Indet 1}
      eval i q ∈ {Det True, Det False, Indet 1}
      eval i r ∈ {Det True, Det False, Indet 1}
    using ranges[of i {1}]
    unfolding domain-def
    by auto
  then show eval i (Cla3 p q r) = Det True
    by (cases eval i p; cases eval i q; cases eval i r) simp-all
qed

proposition valid-in {1, 2} (Cla3 p q r)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range i ⊆ domain {1, 2}
  then have
      eval i p ∈ {Det True, Det False, Indet 1, Indet 2}
      eval i q ∈ {Det True, Det False, Indet 1, Indet 2}
      eval i r ∈ {Det True, Det False, Indet 1, Indet 2}
    using ranges[of i {1, 2}]
    unfolding domain-def
    by auto
  then show eval i (Cla3 p q r) = Det True
    by (cases eval i p; cases eval i q; cases eval i r) auto
qed

proposition ¬ valid-in {1, 2, 3} (Cla3 (Pro ''p'') (Pro ''q'') (Pro ''r''))
proof −
  let ?i = (λs. Indet 1)(''q'' := Indet 2, ''r'' := Indet 3)
  have range ?i ⊆ domain {1, 2, 3}
    unfolding domain-def
    by (simp add: image-subset-iff)
  moreover have eval ?i (Cla3 (Pro ''p'') (Pro ''q'') (Pro ''r'')) = Det False
    by simp
  moreover have Det False ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid-in-def
    by metis
qed
```

You might have noticed that there is a pattern in Δ, Δ_2 and Δ_3. Let us now study that pattern.

Δ can be read as follows: its operand evaluates to a classical value. It is easy to realize that this holds when we have only classical values. It is also clear that

it is not valid already if we allow a single indeterminate value since then the operand might evaluate to that.

Δ_2 can be read as follows: Either p or q evaluates to a classical value, or they evaluate to the same value. It is easy to realize that this holds when we have only one indeterminate value since if none of them evaluate to a classical value then they must both evaluate to the indeterminate one. It is also clear that this does not hold if we allow two indeterminate values since then p and q might respectively evaluate to these two values.

Δ_3 can be read as follows. Either p, q or r evaluates to a classical value, or two of them evaluate to the same value. It is easy to realize that this holds when we only have two indeterminate values, by a similar argument to the one we saw for Δ_2. It is also clear that this does not hold if we allow three indeterminate values, again by a similar argument.

It should be clear that this pattern can be extended as necessary for any number of truth values.

Thus, we now know that in order to check if a formula is valid, we need to consider interpretations with at least as many indeterminate truth values as there are propositional symbols in the formula – otherwise we might have missed an interpretation that falsified the formula. Thus we have found a lower bound on the number of needed indeterminate truth values. In the next section we will find an upper bound on the number of needed indeterminate truth values.

7 A Reduction Theorem

We obtain a reduction theorem by considering the number of propositional symbols in a given formula. Several of the proofs are long — about 250 lines — and are therefore omitted. The theory file for the formalization has the details.

We define a function *props* that returns the set of identifiers for a formula. We then prove that only the propositional symbols in the formula are relevant for the semantics (*relevant-props*).

```
fun props :: fm ⇒ id set
where
  props Truth = {} |
  props (Pro s) = {s} |
  props (Neg' p) = props p |
  props (Con' p q) = props p ∪ props q |
  props (Eql p q) = props p ∪ props q |
  props (Eql' p q) = props p ∪ props q

lemma relevant-props:
  assumes ∀ s ∈ props p. i1 s = i2 s
  shows eval i1 p = eval i2 p
  using assms
  by (induct p) (simp-all, metis)
```

The proof is by induction over formulas (*induct*) followed by simplifications of all cases (*simp-all*) — one case is left over and requires the powerful resolution proof method (*metis*).

We define a function *change-tv* that applies a function to the number in an indeterminate truth value. We then prove that if *f* is an injection then *change-tv f* is also an injection (*change-tv-injection*).

fun *change-tv* :: (*nat* ⇒ *nat*) ⇒ *tv* ⇒ *tv*
where
change-tv f (*Det b*) = *Det b* |
change-tv f (*Indet n*) = *Indet* (*f n*)

lemma *change-tv-injection*: **assumes** *inj f* **shows** *inj* (*change-tv f*)
— Proof omitted

The above proof and the next two proofs are available online.

We define a function *change-int* that takes a function and applies it to an interpretation to get a new interpretation. We then prove that if we replace each indeterminate truth value in an interpretation with another one, then it just changes the result of the formula accordingly (*eval-change*).

definition
change-int :: (*nat* ⇒ *nat*) ⇒ (*id* ⇒ *tv*) ⇒ (*id* ⇒ *tv*)
where
change-int f i ≡ λ*s*. *change-tv f* (*i s*)

lemma *eval-change*:
assumes *inj f*
shows *eval* (*change-int f i*) *p* = *change-tv f* (*eval i p*)
— Proof omitted

We prove that if our formula is valid when we have at least one indeterminate value in our domain for each propositional symbol, then it is valid in general (*valid-in-valid*).

theorem *valid-in-valid*: **assumes** *card U* ≥ *card* (*props p*) **and** *valid-in U p*
shows *valid p*
— Proof omitted

We reformulate the theorem as follows.

theorem *reduce*: *valid p* ⟷ *valid-in* {*1..card* (*props p*)} *p*
using *valid-in-valid transfer*
by *force*

We prove in the final reduction theorem (*reduce*) that we can decide the validity of a given formula by considering as many indeterminacies as the number of propositional symbols in the formula. This also means that the logic is weakened when additional indeterminate truth values are added. For the atomic formula P it is clear that ı suffices. To see this we use the fact that indeterminate truth values are not ordered with respect to truth content. If $[\![P]\!] = ı$ and we replace the truth value with ıı then the truth value is still indeterminate.

8 Entailment — A Case Study

We propose a definition of entailment and verify the results for the case study presented in Section 1.

The following abbreviation *Entail* takes a list of formulas (the assumptions) and a single formula (the conclusion) and returns an equivalent formula with implication and possibly conjunctions.

```
abbreviation (input) Entail :: fm list ⇒ fm ⇒ fm
where
  Entail l p ≡ Imp (if l = [] then Truth else fold Con' (butlast l) (last l)) p

theorem entailment-not-chain:
  ¬ valid (Eql (Entail [Pro ''p'', Pro ''q''] (Pro ''r''))
      (Box ((Imp' (Pro ''p'') (Imp' (Pro ''q'') (Pro ''r'')))))
proof −
  let ?i = (λs. Indet 1)(''r'' := Det False)
  have eval ?i (Eql (Entail [Pro ''p'', Pro ''q''] (Pro ''r''))
      (Box ((Imp' (Pro ''p'') (Imp' (Pro ''q'') (Pro ''r'')))))) = Det False
    by simp
  moreover have Det False ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid-def
    by metis
qed
```

Recall the formulas $P \wedge Q \wedge \neg R$, $P \wedge Q \rightarrow R$ and $R \rightarrow S$. We introduce *B0*, *B1* and *B2* as the corresponding abbreviations.

```
abbreviation (input) B0 :: fm
    where B0 ≡ Con' (Con' (Pro ''p'') (Pro ''q'')) (Neg' (Pro ''r''))

abbreviation (input) B1 :: fm
    where B1 ≡ Imp' (Con' (Pro ''p'') (Pro ''q'')) (Pro ''r'')

abbreviation (input) B2 :: fm
    where B2 ≡ Imp' (Pro ''r'') (Pro ''s'')
```

From *B0* and *B1* we have explosion in classical logic (in the following theorem p is an arbitrary formula in our paraconsistent logic; however, in the proof of the theorem the p in double quotes corresponds to the particular p in *B0* and *B1*).

```
theorem classical-logic-is-not-usable: valid-boole (Entail [B0, B1] p)
  unfolding valid-in-def
proof (rule; rule)
  fix i :: id ⇒ tv
  assume range i ⊆ domain {}
  then have
      i ''p'' ∈ {Det True, Det False}
```

```
    i ''q'' ∈ {Det True, Det False}
    i ''r'' ∈ {Det True, Det False}
   unfolding domain-def
   by auto
 then show eval i (Entail [B0, B1] p) = Det True
   by (cases i ''p''; cases i ''q''; cases i ''r'') simp-all
qed

corollary valid-boole (Entail [B0, B1] (Pro ''r''))
 by (rule classical-logic-is-not-usable)

corollary valid-boole (Entail [B0, B1] (Neg' (Pro ''r'')))
 by (rule classical-logic-is-not-usable)

proposition ¬ valid (Entail [B0, B1] (Pro ''r''))
proof −
 let ?i = (λs. Indet 1)(''r'' := Det False)
 have eval ?i (Entail [B0, B1] (Pro ''r'')) = Det False
   by simp
 moreover have Det False ≠ Det True
   by simp
 ultimately show ?thesis
   unfolding valid-def
   by metis
qed
```

When we consider the full paraconsistent logic, however, everything does not follow. We illustrate this by showing that the negations of p, q and s do not follow. Of these three results that use counterexamples we only include the proof of the last one as they are very similar, but the two others are available in the theory file.

```
proposition ¬ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''p'')))
— Proof omitted

proposition ¬ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''q'')))
— Proof omitted

proposition ¬ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s'')))
proof −
 let ?i = (λs. Indet 1)(''s'' := Det True)
 have eval ?i (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s''))) = Det False
   by simp
 moreover have Det False ≠ Det True
   by simp
 ultimately show ?thesis
   unfolding valid-def
   by metis
qed
```

We do want something to follow – otherwise we would be unable to reason. Indeed something does follow namely r, its negation and s. Of the three results that use proof by cases and the reduction theorem (*reduce*) we only include the proof of the last one as they are very similar, but the two others are available in the theory file.

```
proposition valid (Entail [B0, Box B1, Box B2] (Pro ''r''))
— Proof omitted

proposition valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r'')))
— Proof omitted

proposition valid (Entail [B0, Box B1, Box B2] (Pro ''s''))
proof −
  have {1..card (props (Entail [B0, Box B1, Box B2] (Pro ''s'')))} =
      {1, 2, 3, 4}
    by code-simp
  moreover have valid-in {1, 2, 3, 4}
      (Entail [B0, Box B1, Box B2] (Pro ''s''))
    unfolding valid-in-def
  proof (rule; rule)
    fix i :: id ⇒ tv
    assume range i ⊆ domain {1, 2, 3, 4}
    then have icase:
      i ''p'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''q'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''r'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''s'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      unfolding domain-def
      by auto
    show eval i (Entail [B0, Box B1, Box B2] (Pro ''s'')) = Det True
      using icase
      by (cases i ''p''; cases i ''q''; cases i ''r''; cases i ''s'') simp-all
  qed
  ultimately show ?thesis
    using reduce
    by simp
qed
```

We hence obtain the following results for the agent using the turnstile symbol ($\vdash$) for the entailment given the set of beliefs and rules.

$$\nvdash \neg P \qquad \nvdash \neg Q \qquad \nvdash \neg S$$

$$\vdash R \qquad \vdash \neg R \qquad \vdash S$$

For comparison, due to the catastrophic explosiveness of classical logic, the following results are obtained using classical logic:

$$\vdash \neg P \qquad \vdash \neg Q \qquad \vdash \neg S$$

$$\vdash R \qquad \vdash \neg R \qquad \vdash S$$

9 Related Work

Paraconsistent and/or many-valued logics are occasionally considered in the proof assistant Isabelle. Krauss [21] considers, in a tutorial for Isabelle/HOL, a very small example of a three-valued logic only to illustrate pattern matching in Isabelle/HOL. Brucker, Tuong and Wolff [8] formalize, in Isabelle/HOL, the four-valued logic OCL, which complements the UML software modelling language. Georgescu, Leustean and Preoteasa [13] take an algebraic approach and formalize, in Isabelle/HOL, the theory of pseudo hoops, which is a generalization of the BL-algebra and the many-valued BL-logic [10]. Steen and Benzmüller [39] present a semantic embedding of a many-valued logic in Isabelle/HOL. The truth values are encoded as particular functions and Isabelle/HOL's proof methods are then used directly.

There are several implementations of Dana Scott's Logic of Computable Functions (LCF) [26, 38]. HOLCF [33] extends Isabelle/HOL with ideas from LCF, and allows reasoning about functional programs, including programs that never complete successfully due to errors or non-termination. Regensburger [34] formalizes, in HOLCF, the type of lifted booleans, which consists of true, false and a bottom value. The bottom value of the lifted booleans thus represents a computation of a boolean value that never completes successfully. The type of lifted booleans in HOLCF is also described by Müller, Nipkow, Oheimb and Slotosch [27] as well as by Huffman [18].

We deal with a formalization in a proof assistant of the syntax and semantics of a many-valued paraconsistent logic. Marcos [24] considers another kind of many-valued logic and describes a special computer program in the functional programming language ML. This computer program automatically generates proof tactics to be used by Isabelle. We do not use any computer programs (well, except Isabelle itself, of course). And we do not generate proof tactics. We have ourselves authored all proofs. Furthermore our proofs are in higher-order logic, Isabelle/HOL, which is the default logic in Isabelle. In [24] the default higher-order logic is not used. Instead it is replaced by certain finite-valued logics. This is possible since Isabelle is a generic proof assistant but we do not use this feature at all. More precisely, we formalize the syntax and semantics of the logic, and this is not done in [24]. We can prove theorems in the logic as well as meta-theorems about the logic, but in [24] only theorems in the logic can be proved. Since the logics are rather different it is not possible to compare the efficiency of the two approaches when it comes to proving theorems.

Here, we have only considered the formalization of propositional logic, but the formalization of first-order logic or even higher-order logic is also possible. Several proof systems for classical first-order logic have been proved sound and complete:

- Sequent calculus [6, 35].
- Natural deduction [3, 7, 12, 44].
- Resolution [36].

But even without developing a proof system we can obtain many theorems and meta-theorems by formalizing the syntax and semantics in a proof assistant like Isabelle.

10 Conclusion

In this paper we considered a logic with infinitely many truth values (cf. Section 3 on the syntax and semantics of the logic). Specifically, we investigated how many truth values we need to consider in order to decide if a formula is valid or not. In section 6 we explained that in order to check the validity of a formula with n different propositional symbols we should consider interpretations that use n different indeterminate truth values. The reason was that the formula might be true in any interpretation that uses only $n - 1$ different indeterminate truth values but false in one that uses n of them. We gave concrete examples of such formulas for $n = 1, 2, 3$. The formulas showed a pattern that we argued would generalize to any n. Future work includes a formalization of this argument.

Our main theoretic result is the reduction theorem from Section 7:

proposition *valid p* $\longleftrightarrow$ *valid-in* {*1..card* (*props p*)} *p*
using *reduce* **.**

In the above formulation of the reduction theorem, the right-hand side *valid p* means that a formula p is true in all interpretations — of which there are infinitely many. The left-hand side *valid-in* {*1 ..card* (*props p*)} *p* means that p is true in all the interpretations whose domains are restricted to a finite set consisting of true, false and a number of indeterminate truth values as small as the number of propositional symbols in p. This important result allows us to reduce the infinite number of truth values to a finite number of truth values, for a given formula, and in our case study we use this result. Future work includes further investigations of the practical applications of the reduction theorem.

Using a proof assistant like Isabelle makes it possible to clarify concepts and to catch mistakes. However, we have not found errors in our extended abstract [20]. The formalization of the case study shows the limits since using straightforward proof techniques the results can take up to half a minute for Isabelle to prove using a standard computer. It has been a pleasure to use the Isabelle proof assistant for the formalization of our paraconsistent logic. We plan to make a similar formalization in the Coq proof assistant [4] in order to compare the two systems.

In our extended abstract [20] several of these results were mentioned without proof and we now have precise definitions and formal proofs. The reduction theorem was the most difficult proof and large parts of the 1582-lines theory file for Isabelle2016-1 are omitted in the present paper. After the initial successful proofs we spent a lot of time improving the definitions, theorems and proofs; this also involved discussions with other Isabelle users as well as with our students in order to obtain the most elegant and general results.

Acknowledgements

Thanks to Andreas Halkjær From, Alexander Birch Jensen and John Bruntse Larsen for comments on drafts of the paper. Also thanks to Hendrik Decker and the anonymous reviewers for many constructive comments.

References

1. S. Akama (editor). *Towards Paraconsistent Engineering*. Intelligent Systems Reference Library Volume 110, Springer, 2016.
2. D. Batens, C. Mortensen, G. Priest and J. Van-Bendegem (editors). *Frontiers in Paraconsistent Logic*. Research Studies Press, 2000.
3. S. Berghofer. *First-Order Logic According to Fitting*. Archive of Formal Proofs `http://isa-afp.org/entries/FOL-Fitting.shtml` 2007.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science, Springer, 2004.
5. J. C. Blanchette, S. Böhme and L. C. Paulson. *Extending Sledgehammer with SMT solvers*. Journal of Automated Reasoning, 51(1):109–128, 2013.
6. J. C. Blanchette, A. Popescu and D. Traytel. *Soundness and Completeness Proofs by Coinductive Methods*. Journal of Automated Reasoning, 58(1):149–179, 2017.
7. J. Breitner and D. Lohner. *The Meta Theory of the Incredible Proof Machine*. Archive of Formal Proofs `http://isa-afp.org/entries/Incredible_Proof_Machine.shtml` 2016.
8. A. D. Brucker, F. Tuong and B. Wolff. *Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5*. Archive of Formal Proofs `http://isa-afp.org/entries/Featherweight_OCL.shtml` 2014.
9. W. A. Carnielli, M. E. Coniglio and I. M. L. D'Ottaviano (editors). *Paraconsistency: The Logical Way to the Inconsistent*. Marcel Dekker, 2002.
10. L. C. Ciungu. *Non-commutative Multiple-Valued Logic Algebras*. Springer Monographs in Mathematics, Springer, 2014.
11. H. Decker, J. Villadsen and T. Waragai (editors). *International Workshop on Paraconsistent Computational Logic*. Volume 95 of Roskilde University, Computer Science, Technical Reports, 2002.
12. A. H. From. *Formalized First-Order Logic*. BSc Thesis, Technical University of Denmark, 2017.
13. G. Georgescu, L. Leustean and V. Preoteasa. *Pseudo Hoops*. Archive of Formal Proofs `http://isa-afp.org/entries/PseudoHoops.shtml` 2011.
14. H. Geuvers. *Proof Assistants: History, Ideas and Future*. Sadhana, 34(1):3–25, Springer, 2009.
15. K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. In From Frege to Gödel, J. van Heijenoort, editor, Harvard University Press, 1967.
16. S. Gottwald. *A Treatise on Many-Valued Logics*. Research Studies Press, 2001.
17. S. O. Hansson. *Logic of Belief Revision*. In E. N. Zalta *et al.*, editors, Stanford Encyclopedia of Philosophy, Online Entry `http://plato.stanford.edu/entries/logic-belief-revision/` Winter Edition, 2016.
18. B. Huffman. *Reasoning with Powerdomains in Isabelle/HOLCF*. In O. A. Mohamed, C. Muñoz and S. Tahar, editors, TPHOLs 2008, Emerging Trends Proceedings, pages 45–56. Technical Report, Concordia University, 2008.

19. A. B. Jensen, A. Schlichtkrull and J. Villadsen *First-Order Logic According to Harrison*. Archive of Formal Proofs `http://isa-afp.org/entries/FOL_Harrison.shtml` 2017.
20. A. S. Jensen and J. Villadsen. *Paraconsistent Computational Logic*. In P. Blackburn, K. F. Jørgensen, N. Jones, and E. Palmgren, editors, 8th Scandinavian Logic Symposium: Abstracts, pages 59–61, Roskilde University, 2012.
21. A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. Isabelle Distribution `http://isabelle.in.tum.de/doc/functions.pdf` 2017.
22. L. Lamport. *How to Write a Proof*. American Mathematical Monthly, 102(7):600–608, 1995.
23. L. Lamport. *How to Write a 21st Century Proof*. Journal of Fixed Point Theory and Applications, 11(1):43–63, 2012.
24. J. Marcos. *Automatic Generation of Proof Tactics for Finite-Valued Logics*. In Proceedings of Tenth International Workshop on Rule-Based Programming, pages 91–98, 2009.
25. J. Michaelis and T. Nipkow. *Propositional Proof Systems*. Archive of Formal Proofs `http://isa-afp.org/entries/Propositional_Proof_Systems.shtml` 2017.
26. R. Milner. *Logic for Computable Functions: Description of a Machine Implementation*. Stanford University, 1972.
27. O. Müller, T. Nipkow, D. Oheimb and O. Slotosch. *HOLCF = HOL + LCF*. Journal of Functional Programming 9(2):191–223, 1999.
28. T. Nipkow, L. C. Paulson and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283, Springer, 2002.
29. T. Nipkow and G. Klein. *Concrete Semantics — With Isabelle/HOL*. Springer, 2014. Online Book `http://concrete-semantics.org/` 2017.
30. L. C. Paulson. *A Machine-Assisted Proof of Gödel's Incompleteness Theorems for the Theory of Hereditarily Finite Sets*. Review of Symbolic Logic, 7(3):484–498, 2014.
31. G. Priest, R. Routley and J. Norman (editors). *Paraconsistent Logic: Essays on the Inconsistent*. Philosophia Verlag, 1989.
32. G. Priest, K. Tanaka and Z. Weber. *Paraconsistent Logic*. In E. N. Zalta *et al.*, editors, Stanford Encyclopedia of Philosophy, Online Entry `http://plato.stanford.edu/entries/logic-paraconsistent` Winter Edition, 2016.
33. F. Regensburger. *HOLCF: Higher Order Logic of Computable Functions*. In E. T. Schubert, P. J. Windley and J. Alves-Foss, editors, Higher Order Logic Theorem Proving and Its Applications, pages 293–307. Lecture Notes in Computer Science 971, Springer, 1995.
34. F. Regensburger. *The type of lifted booleans*. Isabelle Distribution `http://isabelle.in.tum.de/library/HOL/HOLCF/Tr.html` 2017.
35. T. Ridge. *A Mechanically Verified, Efficient, Sound and Complete Theorem Prover For First Order Logic*. Archive of Formal Proofs `http://isa-afp.org/entries/Verified-Prover.shtml` 2004.
36. A. Schlichtkrull. *The Resolution Calculus for First-Order Logic*. Archive of Formal Proofs `http://isa-afp.org/entries/Resolution_FOL.shtml` 2016.
37. A. Schlichtkrull and J. Villadsen. *Paraconsistency*. Archive of Formal Proofs `http://isa-afp.org/entries/Paraconsistency.shtml` 2017.
38. D. S. Scott. *A Type-Theoretical Alternative to ISWIM, CUCH, OWHY*. Theoretical Computer Science, 121:411–440, 1993. Annotated version of an unpublished manuscript from 1969.
39. A. Steen and C. Benzmüller. *Sweet SIXTEEN: Automation via Embedding into Classical Higher-Order Logic*. Logic and Logical Philosophy, 25(4): 535–554, 2016.

40. J. Villadsen. *Combinators for Paraconsistent Attitudes*. In P. de Groote, G. Morrill and C. Retoré, editors, Logical Aspects of Computational Linguistics, pages 261–278. Lecture Notes in Computer Science 2099, Springer, 2001.
41. J. Villadsen. *Paraconsistent Assertions*. In G. Lindemann, J. Denzinger, I. J. Timm and R. Unland, editors, Multi-Agent System Technologies, pages 99–113. Lecture Notes in Computer Science 3187, Springer, 2004.
42. J. Villadsen. *A Paraconsistent Higher Order Logic*. In B. Buchberger and J. A. Campbell, editors, Artificial Intelligence and Symbolic Computation, pages 38–51. Lecture Notes in Computer Science 3249, Springer, 2004.
43. J. Villadsen. *Supra-logic: Using Transfinite Type Theory with Type Variables for Paraconsistency*. Logical Approaches to Paraconsistency, Journal of Applied Non-Classical Logics, 15(1):45–58, 2005.
44. J. Villadsen, A. B. Jensen and A. Schlichtkrull. *NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle*. IFCoLog Journal of Logics and their Applications, 4(1): 55–82, 2017.
45. J. Villadsen and A. Schlichtkrull. *Formalization of Many-Valued Logics*. In H. Christiansen, M. D. Jiménez-López, R. Loukanova and L. S. Moss, editors, Partiality and Underspecification in Information, Languages, and Knowledge, chapter 7. Cambridge Scholars Publishing, 2017.
46. Z. Weber. *Paraconsistent Logic*. The Internet Encyclopedia of Philosophy, Online Entry `http://www.iep.utm.edu/para-log` 2017.
47. M. Wenzel. *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz and C. Paulin, editors, Theorem Proving in Higher Order Logics, pages 167–183. Lecture Notes in Computer Science 1690, Springer, 1999.
48. F. Wiedijk. *The Seventeen Provers of the World*. Lecture Notes in Computer Science 3600, Springer, 2006.